**Map<K,V>**

**K is the type of keys maintained by the map and V is the type of mapped values.**

Map cannot contain duplicate keys . Each key can atmost map to one value. Each key value pair is called an entry.A key object is used to retrieve a value at later date.

Methods🡪

((a)) V put(K key, V value) :-- it associates the specified value with the specified key in this map. If the map previously contain a mapping for the key, the old value is replaced by the new value.

b)) V get(Object key) :-- it will return the value to which the specified value is mapped or null if map contains no mapping for the key.

c)) V remove (Object key) :--it removes the key value from the map if the key is present and return the value that is removed (which can be null also if one has inserted null for any key) and if the key not present then also it returns null.

d)) boolean containsKey(Object key) :-- it will return true if the map contains the key.

e)) boolean containValue(Object key) :-- it will return true if the map contain the value.

f)) int size() gives the no of entry in this map,

g))boolean isEmpty() returns true if the map doesnot contain anything,

h))void clear() removes everything from the map

i)) Set<K> keyset() : it will return all the keys present in the map.

j)) Collection <v> values : it will return all the values present in the collection.

k) **void putAll(Map m)** : Puts all the entries from m into this map.

l) **Set entrySet( )** : Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.

m) **boolean equals(Object obj)** : Returns true if obj is a Map and contains the same entries. Otherwise, returns false.

n) **int hashCode( )** : Returns the hash code for the invoking map.

**HashMap**

public class HashMap<K,V>

extends [AbstractMap](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html)<K,V>

implements [Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V>, [Cloneable](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html), [Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

* **Methods inherited from class java.util.**[**AbstractMap**](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html)

[equals](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html#equals(java.lang.Object)), [hashCode](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html#hashCode()), [toString](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html#toString())

* **Methods inherited from class java.lang.**[**Object**](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

[finalize](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#finalize()), [getClass](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#getClass()), [notify](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify()), [notifyAll](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notifyAll()), [wait](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait()), [wait](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait(long)), [wait](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait(long,%20int))

* **Methods inherited from interface java.util.**[**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)

[equals](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html#equals(java.lang.Object)), [hashCode](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html#hashCode())

### Constructor Detail

#### HashMap

* + public HashMap(int initialCapacity,

float loadFactor)

Constructs an empty HashMap with the specified initial capacity and load factor.

****Parameters:****

initialCapacity - the initial capacity

loadFactor - the load factor

****Throws:****

[IllegalArgumentException](http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html) - if the initial capacity is negative or the load factor is nonpositive

#### HashMap

public HashMap(int initialCapacity)

Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

****Parameters:****

initialCapacity - the initial capacity.

****Throws:****

[IllegalArgumentException](http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html) - if the initial capacity is negative.

#### HashMap

public HashMap()

Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

#### HashMap

public HashMap([Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<? extends [K](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html),? extends [V](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)> m)

Constructs a new HashMap with the same mappings as the specified Map. The HashMap is created with default load factor (0.75) and an initial capacity sufficient to hold the mappings in the specified Map.

****Parameters:****

m - the map whose mappings are to be placed in this map

****Throws:****

[NullPointerException](http://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html) - if the specified map is null

**1)How Hashmap works :🡪** Most common interview questions are "How HashMap works in java", "How get and put method of HashMap work internally". Here I am trying to explain internal functionality with an easy example. Rather than going through theory, we will start with example first, so that you will get better understanding and then we will see how get and put function work in java.

Lets take a very simple example. I have a Country class, we are going to use Country class object as key and its capital name(string) as value. Below example will help you to understand, how these key value pair will be stored in hashmap.

**1. Country.java**

[view plainprint?](http://www.java2blog.com/2014/02/how-hashmap-works-in-java.html)

1. **package** org.arpit.java2blog;
2. **public** **class** Country {
4. String name;
5. **long** population;
7. **public** Country(String name, **long** population) {
8. **super**();
9. **this**.name = name;
10. **this**.population = population;
11. }
12. **public** String getName() {
13. **return** name;
14. }
15. **public** **void** setName(String name) {
16. **this**.name = name;
17. }
18. **public** **long** getPopulation() {
19. **return** population;
20. }
21. **public** **void** setPopulation(**long** population) {
22. **this**.population = population;
23. }
25. // If length of name in country object is even then return 31(any random number) and if odd then return 95(any random number).
26. // This is not a good practice to generate hashcode as below method but I am doing so to give better and easy understanding of hashmap.
27. @Override
28. **public** **int** hashCode() {
29. **if**(**this**.name.length()%2==0)
30. **return** 31;
31. **else**
32. **return** 95;
33. }
34. @Override
35. **public** **boolean** equals(Object obj) {
37. Country other = (Country) obj;
38. **if** (name.equalsIgnoreCase((other.name)))
39. **return** **true**;
40. **return** **false**;
41. }
43. }

If you want to understand more about hashcode and equals method of object, you may refer[**hashcode() and equals() method in java**](http://javapostsforlearning.blogspot.in/2014/02/hashcode-and-equals-method-in-java.html)  
  
**2. HashMapStructure.java**(main class)

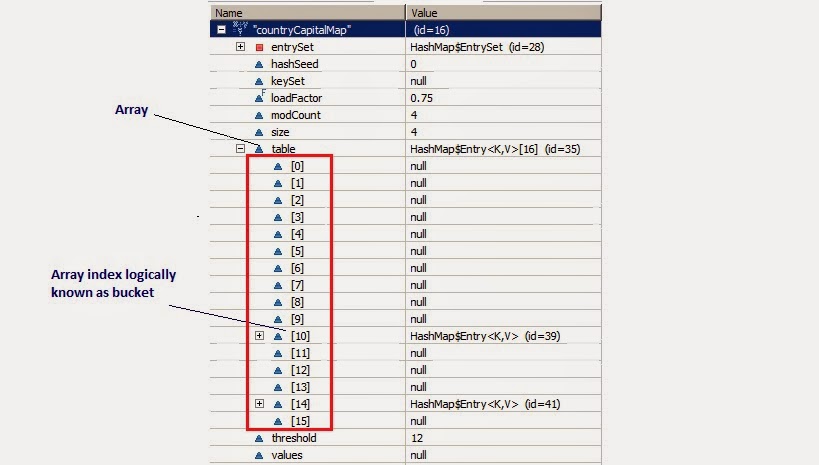
[view plainprint?](http://www.java2blog.com/2014/02/how-hashmap-works-in-java.html)

1. **import** java.util.HashMap;
2. **import** java.util.Iterator;
4. **public** **class** HashMapStructure {
6. /\*\*
7. \* @author Arpit Mandliya
8. \*/
9. **public** **static** **void** main(String[] args) {
11. Country india=**new** Country("India",1000);
12. Country japan=**new** Country("Japan",10000);
14. Country france=**new** Country("France",2000);
15. Country russia=**new** Country("Russia",20000);
17. HashMap<country tring=""> countryCapitalMap=**new** HashMap<country tring="">();
18. countryCapitalMap.put(india,"Delhi");
19. countryCapitalMap.put(japan,"Tokyo");
20. countryCapitalMap.put(france,"Paris");
21. countryCapitalMap.put(russia,"Moscow");
23. Iterator<country> countryCapitalIter=countryCapitalMap.keySet().iterator();//put debug point at this line
24. **while**(countryCapitalIter.hasNext())
25. {
26. Country countryObj=countryCapitalIter.next();
27. String capital=countryCapitalMap.get(countryObj);
28. System.out.println(countryObj.getName()+"----"+capital);
29. }
30. }

33. }
34. </country></country></country>

Now put debug point at line 23 and right click on project->debug as-> java application. Program will stop execution at line 23 then right click on countryCapitalMap then select watch.You will be able to see structure as below.

[image: http://3.bp.blogspot.com/-T4TsTlhwaXU/Uwjj9ypm69I/AAAAAAAACsY/J4-a6rxZLLE/s1600/HashMapStructure1.bmp](http://3.bp.blogspot.com/-T4TsTlhwaXU/Uwjj9ypm69I/AAAAAAAACsY/J4-a6rxZLLE/s1600/HashMapStructure1.bmp)

[[](http://3.bp.blogspot.com/-T4TsTlhwaXU/Uwjj9ypm69I/AAAAAAAACsY/J4-a6rxZLLE/s1600/HashMapStructure1.bmp)](http://3.bp.blogspot.com/-T4TsTlhwaXU/Uwjj9ypm69I/AAAAAAAACsY/J4-a6rxZLLE/s1600/HashMapStructure1.bmp)

Now From above diagram, you can observe following points

1. There is an Entry[] array called table which has size 16.
2. This table stores Entry class's object. HashMap class has a inner class called Entry.This Entry have key value as instance variable. Lets see structure of entry class Entry Structure.

[view plainprint?](http://www.java2blog.com/2014/02/how-hashmap-works-in-java.html)

* 1. **static** **class** Entry **implements** Map.Entry
  2. {
  3. **final** K key;
  4. V value;
  5. Entry next;
  6. **final** **int** hash;
  7. ...//More code goes here
  8. }

1. Whenever we try to put any key value pair in hashmap, Entry class object is instantiated for key value and that object will be stored in above mentioned Entry[](table). Now you must be wondering, where will above created Enrty object get stored(exact position in table). The answer  is, hash code is calculated for a key by calling Hascode() method. This hashcode is used to calculate index for above Entry[] table.
2. Now, If you see at array index 10 in above diagram, It has an Entry object named HashMap$Entry.
3. We have put 4 key-values in hashmap but it seems to have only 2!!!!This is because if two objects have same hashcode, they will be stored at same index. Now question arises how? It stores objects in a form of LinkedList(logically).

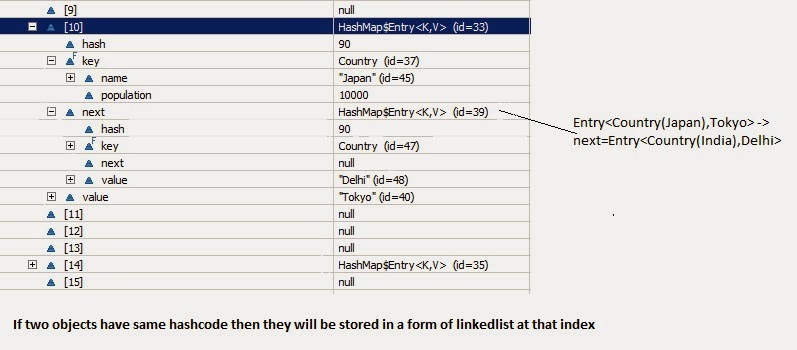
So how hashcode of above country key-value pairs are calculated.

[view plainprint?](http://www.java2blog.com/2014/02/how-hashmap-works-in-java.html)

1. Hashcode **for** Japan = 95 as its length is odd.
2. Hashcode **for** India =95 as its length is odd
3. HashCode **for** Russia=31 as its length is even.
4. HashCode **for** France=31 as its length is even.

Below diagram will explain LinkedList concept clearly.

[image: http://1.bp.blogspot.com/-uBsIhju\_rS0/Uwjnm9ngraI/AAAAAAAACsk/0NoA3f6iHqI/s1600/HashMapStructure2.bmp](http://1.bp.blogspot.com/-uBsIhju_rS0/Uwjnm9ngraI/AAAAAAAACsk/0NoA3f6iHqI/s1600/HashMapStructure2.bmp)

[[](http://1.bp.blogspot.com/-uBsIhju_rS0/Uwjnm9ngraI/AAAAAAAACsk/0NoA3f6iHqI/s1600/HashMapStructure2.bmp)](http://1.bp.blogspot.com/-uBsIhju_rS0/Uwjnm9ngraI/AAAAAAAACsk/0NoA3f6iHqI/s1600/HashMapStructure2.bmp)

 So now if you have good understanding of hashmap structure,Lets go through put and get method.

**Put :**

Lets see implementation of put method:

[view plainprint?](http://www.java2blog.com/2014/02/how-hashmap-works-in-java.html)

1. /\*\*
2. \* Associates the specified value with the specified key in this map. If the
3. \* map previously contained a mapping for the key, the old value is
4. \* replaced.
5. \*
6. \* @param key
7. \*            key with which the specified value is to be associated
8. \* @param value
9. \*            value to be associated with the specified key
10. \* @return the previous value associated with <tt>key</tt>, or <tt>null</tt>
11. \*         if there was no mapping for <tt>key</tt>. (A <tt>null</tt> return
12. \*         can also indicate that the map previously associated
13. \*         <tt>null</tt> with <tt>key</tt>.)
14. \*/
15. **public** V put(K key, V value) {
16. **if** (key == **null**)
17. **return** putForNullKey(value);
18. **int** hash = hash(key.hashCode());
19. **int** i = indexFor(hash, table.length);
20. **for** (Entry<k , V> e = table[i]; e != **null**; e = e.next) {
21. Object k;
22. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
23. V oldValue = e.value;
24. e.value = value;
25. e.recordAccess(**this**);
26. **return** oldValue;
27. }
28. }
30. modCount++;
31. addEntry(hash, key, value, i);
32. **return** **null**;
33. }

now lets understand above code step by step

1. Key object is checked for null. If key is null then it will be stored at table[0] because hashcode for null is always 0.
2. Key object's hashcode() method is called and hash code is calculated. This hashcode is used to find index of array for storing Entry object. It may happen sometimes that, this hashcode function is poorly written so JDK designer has put another function called hash() which takes above calculated hash value as argument.If you want to learn more about hash() function, you can refer [hash and indexFor method in hashmap](http://javapostsforlearning.blogspot.in/2014/02/hash-and-indexfor-method-in-hashmap.html).
3. indexFor(hash,table.length)  is used to calculate exact index in table array for storing the Entry object.
4. As we have seen in our example, if two key objects have same hashcode(which is known as **collision**) then it will be stored in form of linkedlist.So here, we will iterate through our linkedlist.

* If there is no element present at that index which we have just calculated then it will directly put our Entry object at that index.
* If There is element present at that index then it will iterate until it gets Entry->next as null.Then current Entry object become next node in that linkedlist
* What if we are putting same key again, logically it should replace old value. Yes,it will do that.While iterating it will check key equality by calling equals() method(**key.equals(k)**), if this method returns true then it replaces value object with current Entry's value object.

**Get:**

Lets see implementation of get now:

[view plainprint?](http://www.java2blog.com/2014/02/how-hashmap-works-in-java.html)

1. /\*\*
2. \* Returns the value to which the specified key is mapped, or {@code null}
3. \* if this map contains no mapping for the key.
4. \*
5. \* <p>
6. \* More formally, if this map contains a mapping from a key {@code k} to a
7. \* value {@code v} such that {@code (key==null ? k==null :
8. \* key.equals(k))}, then this method returns {@code v}; otherwise it returns
9. \* {@code null}. (There can be at most one such mapping.)
10. \*
11. \* </p><p>
12. \* A return value of {@code null} does not <i>necessarily</i> indicate that
13. \* the map contains no mapping for the key; it's also possible that the map
14. \* explicitly maps the key to {@code null}. The {@link #containsKey
15. \* containsKey} operation may be used to distinguish these two cases.
16. \*
17. \* @see #put(Object, Object)
18. \*/
19. **public** V get(Object key) {
20. **if** (key == **null**)
21. **return** getForNullKey();
22. **int** hash = hash(key.hashCode());
23. **for** (Entry<k , V> e = table[indexFor(hash, table.length)]; e != **null**; e = e.next) {
24. Object k;
25. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))
26. **return** e.value;
27. }
28. **return** **null**;
29. }

As you got the understanding on put functionality of hashmap. So to understand get functionality is quite simple. If you pass any key to get value object from hashmap.

1. Key object is checked for null. If key is null then value of Object resides at table[0] will be returned.
2. Key object's hashcode() method is called and hash code is calculated.
3. indexFor(hash,table.length)  is used to calculate exact index in table array using generated hashcode for getting the Entry object.
4. After getting index in table array, it will iterate through linkedlist and check for key equality by calling equals() method and if it returns true then it returns the value of Entry object else returns null.

**Key points to Remeber:**

* HashMap has a inner class called Entry which stores key-value pairs.
* Above Entry object is stored in Entry[ ](Array) called table
* An index of table is logically known as bucket and it stores first element of linkedlist
* Key object's hashcode() is used to find bucket of that Entry object.
* If two key object 's have same hashcode , they will go in same bucket of table array.
* Key object 's equals() method is used to ensure uniqueness of key object.
* Value object  's equals() and hashcode() method is not used at all

# [How HashMap works in Java](http://javarevisited.blogspot.in/2011/02/how-hashmap-works-in-java.html)

HashMap in Java works on hashing principle. It is a data structure which allows us to store object and retrieve it in constant time O(1) provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling put(key, value) method of HashMap and retrieved by calling get(key) method. When we call put method, hashcode() method of key object is called so that hash function of map can find a bucket location to store value object, which is actually index of internal array, known as table. HashMap internally store mapping in form of Map.Entry object which contains both key and value object. When you want to retrieve the object, you call get() method and again pass key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little tricky when collisions occurs. Since internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is formed at that bucket location and new entry is stored as next node. If we try to retrieve object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keep comparing entry's key object with passed key using equals() and when it return true, Map returns corresponding value. Since searching in lined list is O(n) operation, in worst case hash collision reduce a map to linked list. This issue is recently addressed in Java 8 by replacing linked list to tree to search in O(logN) time. By the way, you can easily verify how HashMap work by looking at code of HashMap.java in your Eclipse IDE, if you know [how to attach source code of JDK in Eclipse](http://javarevisited.blogspot.com/2012/12/how-to-attach-source-in-eclipse-Jar-JDK-debugging.html).  
  
  
How HashMap works in Java or sometime how get method work in HashMap is a very common question on Java interviews now days. Almost everybody who worked in Java knows about HashMap, where to use HashMap and difference between Hashtable and HashMap then why this interview question becomes so special? Because of the depth it offers. It has become very popular Java interview question in almost any senior or mid-senior level Java interviews. Investment banks mostly prefer to ask this question and some time even ask you to implement your own HashMap based upon your coding aptitude. Introduction of [ConcurrentHashMap](http://javarevisited.blogspot.co.uk/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) and other concurrent collections has also made this questions as starting point to delve into more advanced feature. let's start the journey.

## How HashMap Internally Works in Java

Questions start with simple statement :

**Have you used HashMap before**or**What is HashMap? Why do you use it**

Almost everybody answers this with yes and then interviewee keep talking about common facts about HashMap like HashMap accept null while Hashtable doesn't, [HashMap is not synchronized](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html), HashMap is fast and so on along with basics like its stores key and value pairs etc. This shows that person has used HashMap and quite familiar with the functionality it offers, but interview takes a sharp turn from here and next set of follow-up questions gets more detailed about fundamentals involved with HashMap in Java . Interviewer strike back with questions like :

**Do you Know how HashMap works in Java** or **How does get () method of HashMap works in Java**

And then you get answers like,  I don't bother its standard Java API, you better look code on Java source or Open JDK; I can find it out in Google at any time etc. But some interviewee definitely answer this and will say **HashMap works on principle of hashing**, we have put(key, value) and get(key) method for storing and retrieving Objectsfrom HashMap. When we pass Key and Value object  to put() method on Java HashMap, HashMap implementation calls [hashCode method](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html)on Key object and applies returned hashcode into its own hashing function to find a bucket location for storing Entry object, important point to mention is that HashMap in Java stores both key and value object asMap.Entry in bucket which is essential to understand the retrieving logic. If people fails to recognize this and say it only stores Value in the bucket they will fail to explain the retrieving logic of any object stored in Java HashMap . This answer is very much acceptable and does make sense that interviewee has fair bit of knowledge on how hashing works and how HashMap  works in Java. But this is just start of story and confusion increases when you put interviewee on scenarios faced by Java developers on day by day basis. Next question could be about collision detection and collision resolution in Java HashMap  e.g.

**What will happen if two different objects have same hashcode?**

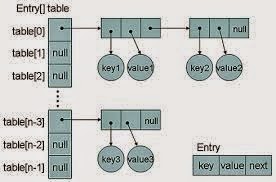
Now from here onwards real confusion starts, Some time candidate will say that since hashcode is equal, both objects are equal and HashMap  will throw exception or not store them again etc, Then you might want to remind them about [equals() and hashCode() contract](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html)that two unequal object in Java can have same hash code. Some will give up at this point and few will move ahead and say "Since hashcode is same, bucket location would be same and collision will occur in HashMap, Since HashMap use LinkedList to store object, this entry (object of Map.Entry comprise key and value )  will be stored in [LinkedList](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html). Great this answer make sense though there are many collision resolution methods available  like linear probing and chaining, this is simplest and HashMap in Java does follow this. But story does not end here and interviewer asks

**How will you retrieve Value object  if two Keys will have same hashcode?**

[how HashMap works internally in Java](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)Interviewee will say we will call get() method and then HashMap uses Key Object's hashcode to find out bucket location and retrieves Value object but then you need to remind him that there are two Value objects are stored in same bucket , so they will say about [traversal in LinkedList](http://javarevisited.blogspot.sg/2010/10/how-do-you-find-length-of-singly-linked.html)until we find the value object , then you ask *how do you identify value object because you don't  have value object to compare* ,Until they know that HashMap  stores both Key and Value in LinkedList node or as Map.Entry they won't be able to resolve this issue and will try and fail.

But those bunch of people who remember this key information will say that after finding bucket location , we will **call keys.equals() method** to identify correct node in LinkedList and return associated value object for that key in Java HashMap . Perfect this is the correct answer.

In many cases interviewee fails at this stage because they get confused between[hashCode()](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) and equals(**)** or keys and values object in Java HashMap  which is pretty obvious because they are dealing with the hashcode() in all previous questions and equals() come in picture only in case of retrieving value object from HashMap in Java. Some good developer point out here that using immutable, [final object](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) with proper equals() and hashcode() implementation would act as perfect Java HashMap  keys and**improve performance of Java HashMap  by reducing collision**. Immutability*also allows caching there hashcode of different keys* which makes overall retrieval process very fast and suggest that [String](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html)and various wrapper classes e.g. Integer very good keys in Java HashMap.

[](http://4.bp.blogspot.com/-adRczhctozE/VD_eimhTQbI/AAAAAAAACCg/lfA1G5GZXyM/s1600/How+HashMap+works+in+Java+(1).jpg)  
  
  
Now if you clear this entire Java HashMap interview,  You will be surprised by this very interesting question "**What happens On HashMap in Java if the size of the HashMap  exceeds a given threshold defined by load factor ?"**. Until you know how HashMap  works exactly you won't be able to answer this question. If the size of the Map exceeds a given threshold defined by load-factor e.g. if load factor is .75 it will act to re-size the map once it filled 75%. Similar to other collection classes like [ArrayList](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html),  Java HashMap re-size itself by creating a new bucket array of size twice of previous size of HashMap , and then start putting every old element into that new bucket array. This process is called rehashing because it also applies hash function to find new bucket location.

If you manage to answer this question on HashMap in Java you will be greeted by **"do you see any problem with resizing of HashMap  in Java"**, you might not be able to pick the context and then he will try to give you hint about multiple thread accessing the Java HashMap and potentially looking for **race condition on HashMap  in Java**.

So the answer is Yes there is potential [race condition](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) exists while resizing HashMap in Java, if two [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html)at the same time found that now HashMap needs resizing and they both try to resizing. on the process of resizing of HashMap in Java , the element in bucket which is stored in linked list get reversed in order during there migration to new bucket because Java HashMap  doesn't append the new element at tail instead it append new element at head *to avoid tail traversing*. If race condition happens then you will end up with an infinite loop. Though this point you can potentially argue that what the hell makes you think to use HashMap  in multi-threaded environment to interviewer :)

## [Update] Improvements in Java 8

As part of the work for [JEP 180](http://openjdk.java.net/jeps/180), there is a performance improvement for HashMap objects where there are lots of collisions in the keys by using balanced trees rather than linked lists to store map entries. The principal idea is that **once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a linked list of entries to a balanced tree. In the case of high hash collisions, this will improve worst-case performance from O(n) to O(log n)**.

Basically when a bucket becomes too big (**currently: TREEIFY\_THRESHOLD = 8**), HashMap dynamically replaces it with an ad-hoc implementation of tree map. This way rather than having pessimistic O(n) we get much better O(log n).

Bins (elements or nodes) of TreeNodes may be traversed and used like any others, but additionally support faster lookup when overpopulated. However, since the vast majority of bins in normal use are not overpopulated, checking for existence of tree bins may be delayed in the course of table methods.

Tree bins (i.e., bins whose elements are all TreeNodes) are ordered primarily by hashCode, but in the case of ties, if two elements are of the same “class C implements Comparable<C>“, type then their compareTo() method is used for ordering.

Because TreeNodes are about twice the size of regular nodes, we use them only when bins contain enough nodes. And when they become too small (due to removal or resizing) they are converted back to plain bins (**currently: UNTREEIFY\_THRESHOLD = 6**). In usages with well-distributed user hashCodes, tree bins are rarely used.

I hope, i have correctly communicated my thoughts by this article. If you find any difference or need any help in any point, please drop a comment.

\*\*\*[**get**](http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html#get(java.lang.Object))([Object](http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html) key)   
          Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

\*\*\*public V put(K key, V value) {  
//Some code  
}

will return  
  
1.  null , if key is unique and added to the map  
2.  Old Value of the key , if key is duplicate

\*\*\*There can be no duplicate in key but value may be duplicate. And only one null key is allowed.

**LINKEDHASHMAP<K,V>**

It is a subclass of HashMap. It has same functionality as HashMap except it maintains insertion order. It is a combination of HashTable and LinkedList.

This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.

Entry of LinkedHashMap looks like this🡪

static class Entry<K, V> {

K key;

V value;

Entry<K,V> next;

Entry<K,V> before, after; //For maintaining insertion order

public Entry(K key, V value, Entry<K,V> next){

this.key = key;

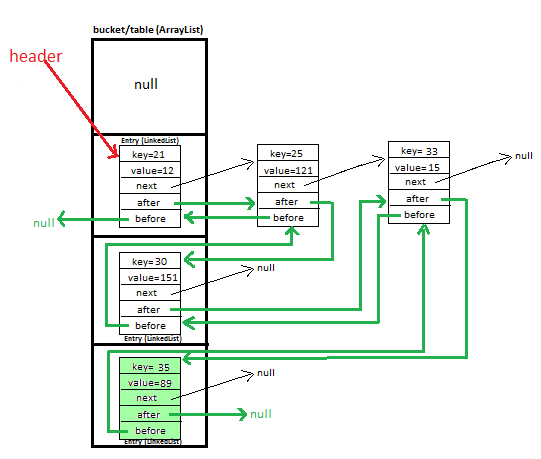
this.value = value;

this.next = next;

}

}

***Custom LinkedHashMap >***



[](http://pinterest.com/pin/create/button/?url=http://www.javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html&media=https://lh4.googleusercontent.com/5M89iub9wNrKR-jmctcmLtGIHiy0lbqL5Oz18AvQyuHdFXGDC4qKgCVZO73WPujj3qqquh_WXlKp7zem4vCfEynJzcN-t0a_wa_mSxDO62G2C-YgEvUhQRBdXdWvpBIhDDM0BxE&description=LinkedHashMap%20Custom%20implementation%20in%20java%20-%20How%20LinkedHashMap%20works%20internally%20with%20diagrams%20and%20full%20program)

This is very **important and** **trending** topic. In this post i will be explaining **LinkedHashMap** custom implementation with diagrams which will help you in **visualizing** the LinkedHashMap implementation.

I will be explaining how we will **put** and **get** key-value pair in HashMap by overriding-

>**equals** method - helps in checking equality of entry objects.

>**hashCode** method - helps in finding bucket’s index on which data will be stored.

We will maintain **bucket (**[**ArrayList**](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)**)** which will store **Entry (**[**LinkedList**](http://javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html)**).**

Most salient feature of **LinkedHashMap** is that it **maintains insertion order** of key-value pairs. We will maintain [doubly Linked List](http://javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html) for doing so.

While our [HashMap](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html) didn’t maintained insertion order.

***Entry<K,V>***

We store key-value pair by using **Entry<K,V>**

By using,  **Entry<K,V> before, after -**  we keep track of newly added entry in LinkedHashMap, which helps us in **maintaining insertion order**.

Entry contains

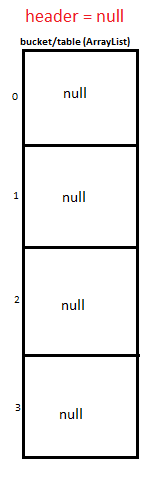
* K **key,**
* V **value,**
* Entry<K,V> **next**(i.e. next entry on that location of bucket),
* Entry<K,V> **before** and
* Entry<K,V> **after**

|  |
| --- |
| **static** **class** Entry<K, V> {         K key;         V value;         Entry<K,V> next;  **Entry<K,V> before, after** ;    **public** Entry(K key, V value, Entry<K,V> next){  **this**.key = key;  **this**.value = value;  **this**.next = next;         }     } |

*Putting* ***5 key-value pairs in custom LinkedHashMap (step-by-step)>***

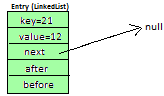
I will explain you the whole concept of **LinkedHashMap** by putting **5 key-value pairs in HashMap.**

**Initially,** we have bucket of **capacity=4.** (all indexes of bucket i.e. 0,1,2,3 are pointing to null)

****

**Let’s put first key-value pair in LinkedHashMap-**

**Key=21, value=12**

**newEntry Object** will be formed like this >

We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 21%4= 1**.

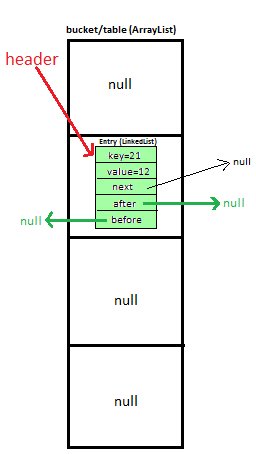
So, **1** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st** index as it is pointing to null we will **put our newEntry object there**.

**Additionally, for maintaining insertion order-**

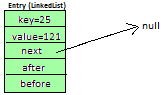
Update **header**, it will start pointing to **newEntry object**

At completion of this step, our HashMap will look like this-

****

**Let’s put second key-value pair in LinkedHashMap-**

**Key=25, value=121**

**newEntry Object** will be formed like this >

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 25%4= 1.**

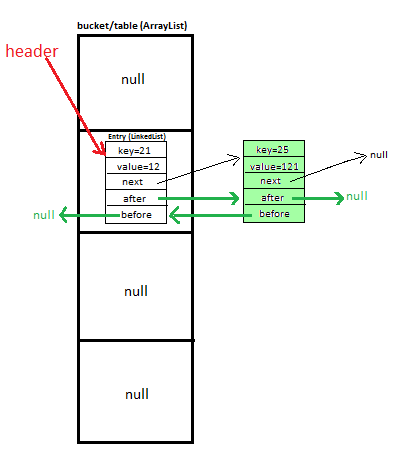
**So, 1** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st** index, it contains **entry with key=21**, we will compare two keys(i.e. **compare 21 with 25** by using **equals method**), as **two keys are different** we check whether entry with key=21’s **next is null or not**, **if next is null** we will **put** our **newEntry object** on **next.**

**Additionally, for maintaining insertion order-**

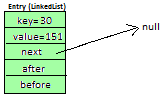
Update **header.after**, it will start pointing to **newEntry object** (i.e make Entry with key=21’s after point to **newEntry object**], and also make  **newEntry object’s** before point to header(Entry with key=21’)

At completion of this step our HashMap will look like this-

****

**Let’s put third key-value pair in HashMap-**

**Key=30, value=151**

**newEntry Object** will be formed like this >

We will calculate hash by using our **hash(K key)** method **-** in this case it returns

**key/capacity= 30%4= 2**.

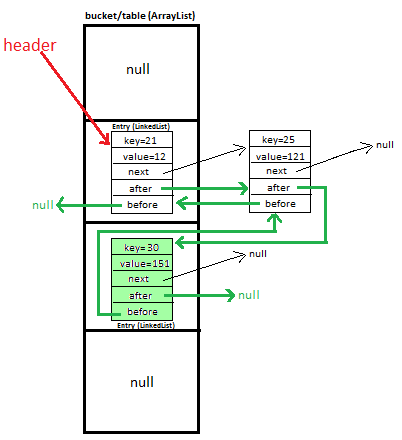
So, **2** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **2nd** index as it is pointing to null we will **put our newEntry object there**.

**Additionally, for maintaining insertion order-**

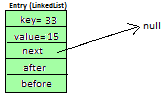
Update doubly linked list ’s **after and before.**

At completion of this step, our HashMap will look like this-



**Let’s put fourth key-value pair in LinkedHashMap-**

**Key=33, value=15**

Entry Object will be formed like this >

We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 33%4= 1,**

**So, 1** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st** index -

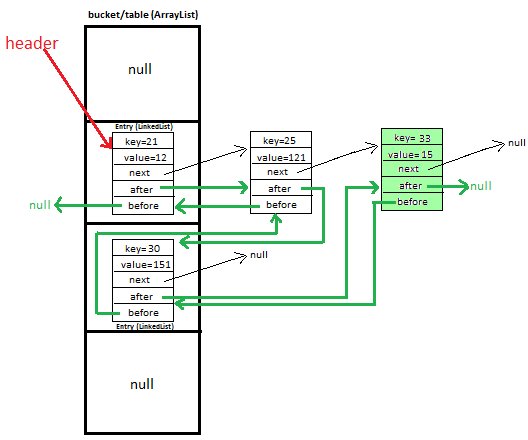
**>**it contains **entry with key=21**, we will **compare** two keys (i.e. **compare 21 with 33** by using **equals method**, as **two keys are different,**proceed to next  of **entry with key=21 (**proceed only if **next is not null).**

**>**now, next contains **entry with key=25**, we will **compare** two keys (i.e. **compare 25 with 33** by using **equals method**, as **two keys are different,**now **next of entry with key=25** is pointing to **null** so we won’t proceed **further,** we will **put our newEntry object on next.**

**Additionally, for maintaining insertion order-**

Update doubly linked list’s **after and before** (for maintaining insertion order)

At completion of this step our HashMap will look like this-

****

LinkedHashMap is an implementation of [java.util.Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html) interface with predictable iteration order (the order of insertion) i.e. LinkedHashMap will iterate in the order in which the entries were put into the map. LinkedHashMap will iterate in the order in which the entries were put into the map.

Performance of LinkedHashMap is slightly below than that of HashMap, due to the added expense of maintaining the linked list.

If performance is critical but ordering is not, go for HashMap. If sorting is important, for example showing values in a table, sorted in alphabetical order then go for TreeMap. If a value is added in the TreeMap or removed from the TreeMap, TreeMap will make sure the table is still sorted alphabetically. If the order of insertion is important, for example showing values in a shopping cart, go for LinkedHashMap.

A LinkedHashMap is a combination of hash table and linked list. It has a predictable iteration order (a la linked list), yet the retrieval speed is that of a HashMap. The order of the iteration is determined by the insertion order, so you will get the key/values back in the order that they were added to this Map. You have to be a bit careful here, since re-inserting a key does not change the original order.

**\*\*\***This class provides all of the optional Map operations, and permits null elements. Like HashMap, it provides constant-time performance for the basic operations (add, contains and remove), assuming the hash function disperses elements properly among the buckets. Performance is likely to be just slightly below that of HashMap, due to the added expense of maintaining the linked list, with one exception: Iteration over the collection-views of a LinkedHashMap requires time proportional to the size of the map, regardless of its capacity. Iteration over a HashMap is likely to be more expensive, requiring time proportional to its capacity.

**What happens when one key is replaced by new key in LinkedHashMap?**

The exception is that when a key is reinserted, it appears in the order in which it was first inserted to the list.

**TREEMAP**

TreeMap is sorted according to the natural sorting order of the keys.It is same as HashMap instead it maintains ascending order. It is a red-black tree based NavigableMap implementation.

**Constructors🡪**

* 1. Public TreeMap() : it will create a new empty treemap using natural sorting of the keys. All key inserted into the Map must implement Comparable interface else it will throw ClassCastException.
  2. Public TreeMap(Map m),
  3. Public TreeMap(SortedMap m)
  4. Public TreeMap(Comparator c) : it will create empty TreeMap but the default sorting order will be determined by Comparator object c.

**Methods🡪**

TreeMap has some extra methods which are not there in Map interface. These are :

1. V firstKey() :🡪 it will return first key of the TreeMap.
2. V lastKey() :🡪 it will return the last key of the TreeMap.
3. V ceilingKey(K k) :🡪 it will return the first >= value of the passed parameter k.
4. V floorKey(K k) :🡪 it will return the first <= value of the passed parameter k.
5. NavigableSet<V> descendingKeySet() :🡪 it will return a NavigableSet which will contain all the keys in descending order.
6. NavigableMap<K,V> descendingMap():🡪 it will return a NavigableMap which will contain all the map objects where keys are in descending order.
7. SortedMap<K,V> headMap(K k, boolean inclusive ) :🡪 it will return the SortedMap with keys less than k. if inclusive is true then it will include the entry with key equal to value k. By defaulf inclusive is false.
8. SortedMap<K,V>tailMap(K k,boolean inclusive) :🡪 it will return the SortedMap with keys greater than equal to k. by default inclusive is true. If we make it false then the result will not contain the equal key value.
9. NavigableMap <K,V>subMap(K k1,boolean inclusive1, K k2,boolean inclusive2) :🡪

It returns the subMap between k1 and k2. And inclusion is defined by inclusive1 and inclusive2. By default inclusive1 is true and inclusive2 is false.

10)pollFirstEntry() :🡪 it removes the first entry and returns the same

In TreeMap whenever we will add one key value object then the key object should have been derived from Comparable Interface . Else we can use Comparator interface implemented class object to pass inside TreeMap constructor.

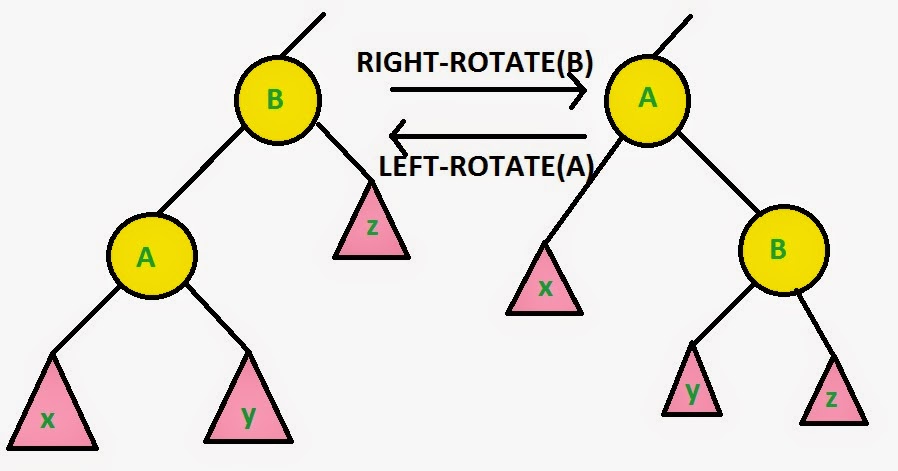
Whenever we call put() method in TreeMap object then compareTo() method gets called in the treemap if no compactor object is passed in the constructor else

the compare() method of the Comparator class gets called.

# How TreeMap Works In Java : 10 TreeMap Java Interview Questions

One of the commonly asked  [Java Developer Interview Questions](http://javahungry.blogspot.com/2013/06/top-25-most-frequently-asked-core-java.html) is How TreeMap works in java or internal implementation of TreeMap. The problem with the candidate that he/she knows what TreeMap/TreeSet do but not How ?  We have already discussed  other popular  java developer interview questions like [how hashmap works](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html) in java , [how hashset works](http://javahungry.blogspot.com/2013/08/how-sets-are-implemented-internally-in.html) in java , what is the [difference between hashmap and hashtable](http://javahungry.blogspot.com/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html). So let us discuss the working of TreeMap.  
  
**What is a Tree Map ?**  
  
Treemap class is like HashMap which stores key- value pairs . The major difference is that Treemap  sorts  
the key in ascending order.  
  
According to [Java doc](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html)  :  
  
Treemap is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.  
This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

**How TreeMap works in java**?  
  
  
TreeMap is a Red-Black tree based NavigableMap implementation.In other words , it sorts the TreeMap object keys using Red-Black tree algorithm.  
  
  
**So we learned that TreeMap uses Red Black tree algorithm internally to sort the elements.**   
  
  
Red Black algorithm is a complex algorithm . We should read the pseudo code of Red Black algorithm in order to understand the internal implementation .  
  
Red Black tree has the following properties :  
  
1. As the name of the algorithm suggests ,color of every node in the tree is either red or black.  
  
2. Root node must be Black in color.  
  
3. Red node can not have a red color neighbor node.  
  
4. All paths from root node to the null should consist the same number of black nodes .  
  
  
**Rotation in Red Black Tree :**

[](http://1.bp.blogspot.com/-4UOANsOx7jo/U6FqoIG6Z2I/AAAAAAAAArc/7f1hAJv-AOA/s1600/how+treemap+works+in+java.jpg)

Rotations maintains the inorder ordering of the keys(x,y,z).  
A rotation can be maintained in O(1) time.  
  
You can find more about the red black tree algorithm [here](http://www.csanimated.com/animation.php?t=Red-black_tree)  
  
**Interviewer : Why and when we use TreeMap ?**  
  
We need TreeMap  to get the sorted list of keys in ascending order.  
  
**Interviewer : What is the runtime performance of the get() method in TreeMap and HashMap  ,where n represents the number of elements ?**  
  
According to [TreeMap Java doc](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html),  
  
TreeMap implementation provides guaranteed log(n) time cost for the containsKey,get,put and remove operations.  
  
According to [HashMap Java doc](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) :  
  
HashMap implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets.  
  
One liner : TreeMap : log(n)   HashMap : Constant time performance assuming elements disperses properly  
  
**Interviewer : What is "natural ordering" in TreeMap ?**  
  
"Natural" ordering is the ordering implied by the implementation of the Comparable interface by the objects used as keys in the TreeMap. Essentially, RBTree must be able to tell which key is smaller than the other key, and there are two ways to supply that logic to the RBTree implementation:  
  
1.Implement Comparable interface in the class(es) used as keys to TreeMap, or  
2.Supply an implementation of the Comparator that would do comparing outside the key class itself.

Natural ordering is the order provided by the Comparable interface .If somebody puts the key  that do not implement natural order then it will throw ClassCastException.  
  
  
**Interviewer : Why do we need TreeMap when we have sortedMap ?**  
sortedMap is a interface and TreeMap is the class implementing it .As we know one can not create objects of the interface . Interface tells us which methods a sortedMap implementation should provide .TreeMap is such an implementation.

**Interviewer : Which data structure you will prefer in your code : HashMap or TreeMap ?**  
  
HashMap is faster while  TreeMap is sorted .Thus we choose them according to their advantage.  
  
If you do not want to sort the elements but just to insert and retrieve the elements then use HashMap .  
  
But if you want to maintain the  order of the elements then TreeMap should be preferred because the result is alphabetically sorted .While iterating HashMap there is no ordering of the elements ,on the other hand , TreeMap iterates in the natural key order.

**Interviewer : What happens if the TreeMap is concurrently modified while iterating the elements ?**  
  
The iterator *fails fast* and quickly if structurally modified at any time after the iterator is created (in any way except through the iterator's own remove method ). We already discussed the [difference between Fail-fast and Fail safe iterators](http://javahungry.blogspot.com/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html) .  
  
**Interviewer : Which copy technique (deep or shallow ) is used by the TreeMap clone() method ?**  
  
According to [docjar](http://www.docjar.com/html/api/java/util/TreeMap.java.html) , clone() method returns the shallow copy of the TreeMap instance . In shallow copy object B points to object A location in memory . In other words , both object A and B are sharing the same elements .The keys and values  themselves are not cloned .  
  
**Interviewer : Why  java's  treemap does not allow an initial size ?**  
  
HashMap reallocates its internals as the new one gets inserted while TreeMap does not reallocate nodes on adding new ones. Thus , the size of the TreeMap  dynamically increases if needed , without shuffling the internals. So it is meaningless to set the initial size of the TreeMap .

\*\*\*In TreeMap the put and get method does not depends upon equal and hashcode. It depends upon compareTo() method of the key object or the compare() method defined by the Comparator object associated with the TreeMap. a TreeSet instance performs all element comparisons using its compareTo (or compare) method

equals() and hashCode do not come into the picture when dealing when TreeSet and TreeMap. However, it is a good practice to override them properly, should you use this object as a key for HashMap (for example) in the future.

╔══════════════╦═════════════════════╦═══════════════════╦══════════════════════╗

║ Property ║ HashMap ║ TreeMap ║ LinkedHashMap ║

╠══════════════╬═════════════════════╬═══════════════════╬══════════════════════╣

║ ║ no guarantee order ║ sorted according ║ ║

║ Order ║ will remain constant║ to the natural ║ insertion-order ║

║ ║ over time ║ ordering ║ ║

╠══════════════╬═════════════════════╬═══════════════════╬══════════════════════╣

║ Get/put ║ ║ ║ ║

║ remove ║ O(1) ║ O(log(n)) ║ O(1) ║

║ containsKey ║ ║ ║ ║

╠══════════════╬═════════════════════╬═══════════════════╬══════════════════════╣

║ ║ ║ NavigableMap ║ ║

║ Interfaces ║ Map ║ Map ║ Map ║

║ ║ ║ SortedMap ║ ║

╠══════════════╬═════════════════════╬═══════════════════╬══════════════════════╣

║ ║ ║ ║ ║

║ Null ║ allowed ║ only values ║ allowed ║

║ values/keys ║ ║ ║ ║

╠══════════════╬═════════════════════╩═══════════════════╩══════════════════════╣

║ ║ Fail-fast behavior of an iterator cannot be guaranteed ║

║ Fail-fast ║ impossible to make any hard guarantees in the presence of ║

║ behavior ║ unsynchronized concurrent modification ║

╠══════════════╬═════════════════════╦═══════════════════╦══════════════════════╣

║ ║ ║ ║ ║

║Implementation║ buckets ║ Red-Black Tree ║ double-linked ║

║ ║ ║ ║ buckets ║

╠══════════════╬═════════════════════╩═══════════════════╩══════════════════════╣

║ Is ║ ║

║ synchronized ║ implementation is not synchronized ║

╚══════════════╩════════════════════════════════════════════════════════════════╝

**HASHTABLE**

Hashtable is deprecated from 1.2 v. It is similar to HashMap except it is synchronized.

It does not allow null key and values. It is not used now .Now if you want threadsafe HashMap you should use ConcurrentHashMap.

**Constructors :🡪** 1)HashTable() creates with initial capacity 11 and load factor .75

2) HashTable(int initialCapacity), 3)HashTable(int initialCapacity, float loadfactor) , 4)HashTable(Map m).

**Methods:🡪** HashTable has method named contains() which is identical to containsValue().

HashTable put() method returns previous value associated with key same as HashMap.

HashTable also has keys() method which returns enumeration of keys. Values() method also are there which returns collection of values. Elements() method will return enumeration of elements.

**Comparative Study**

HashTable Working is exactly same as HashMap. Both use hashing technique to store unique key values.

|  |  |
| --- | --- |
| HashTable | HashMap |
| Synchronized and threadsafe | Not threadsafe and not not synchronized |
| Does not allow any null key or value | It allow one null key and multiple null values |
| Legacy class | Introduced in 1.2 v |
| Slow | Faster and use less memory |
| Traversed by enumerator and iterator | Traversed by only iterator |
| Enumerator in Hashtable is not fail-fast | Iterator in hashMap is fail-fast |
| Inherits Abstract Map class | Inherits Dictionary class |

**Difference between HashMap and HashTable / HashMap vs HashTable**    
  
**1. Synchronization or Thread Safe :**  This is the most important difference between two . HashMap is non synchronized and not thread safe.On the other hand, HashTable is thread safe and synchronized.  
When to use HashMap ?  answer is if your application do not require any multi-threading task, in other words hashmap is better for non-threading applications. HashTable should be used in multithreading applications.   
  
**2. Null keys and null values :**  Hashmap allows one null key and any number of null values, while Hashtable do not allow null keys and null values in the HashTable object.  
  
  
  
**3. Iterating the values:**  Hashmap object values are iterated by using iterator .HashTable is the only class other than vector which uses enumerator to iterate the values of HashTable object.

**4.  Fail-fast iterator**  : The iterator in Hashmap is fail-fast iterator while the enumerator for Hashtable is not.  
According to [Oracle Docs](http://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html),  if the Hashtable is structurally modified at any time after the iterator is created in any way except the iterator's own remove method , then the iterator will throw ConcurrentModification Exception.  
Structural modification means adding or removing elements from the Collection object (here hashmap or hashtable) . Thus the enumerations returned by the Hashtable keys and elements methods are not fail fast.We have already explained the[difference between iterator and enumeration](http://javahungry.blogspot.com/2013/06/difference-between-iterator-and-enumeration-collections-java-interview-question-with-example.html).  
  
  
**5. Performance :**  Hashmap is much faster and uses less memory than Hashtable as former is unsynchronized . Unsynchronized objects are often much better in performance in compare to synchronized  object like Hashtable in single threaded environment.  
  
**6. Superclass and Legacy :**  Hashtable is a subclass of Dictionary class which is now obsolete in Jdk 1.7 ,so ,it is not used anymore. It is better off externally synchronizing a HashMap or using a ConcurrentMap implementation (e.g ConcurrentHashMap).HashMap is the subclass of the AbstractMap class. Although Hashtable and HashMap has different superclasses but they both are implementations of the *"Map"*  abstract data type.

**Similarities Between HashMap and Hashtable**  
  
**1. Insertion Order :**   Both HashMap and Hashtable  does not guarantee that  the order of the map will remain constant over time. Instead use LinkedHashMap, as the order remains constant over time.  
  
**2. Map interface :**   Both HashMap and Hashtable implements Map interface .  
  
**3. Put and get method :**  Both HashMap and Hashtable provides constant time performance for put and get methods assuming that the objects are distributed uniformly across the bucket.  
  
**4. Internal working :**  Both HashMap and Hashtable works on the Principle of Hashing . We have already discussed [how hashmap works in java](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html) .  
  
  
**When to use HashMap and Hashtable?**  
  
*1. Single Threaded Application*  
  
HashMap should be preferred over Hashtable for the non-threaded applications. In simple words , use HashMap in unsynchronized or single threaded applications .  
  
*2. Multi Threaded Application*  
  
We should avoid using Hashtable, as the class is now obsolete in latest Jdk 1.8 . Oracle has provided a better replacement of Hashtable named ConcurrentHashMap. For multithreaded  application prefer ConcurrentHashMap instead of Hashtable.  
  
  
**Recap  : Difference between HashMap and Hashtable in Java** 

|  |  |  |
| --- | --- | --- |
|  | **HashMap** | **Hashtable** |
|  |  |  |
| Synchronized | No | Yes |
|  |  |  |
| Thread-Safe | No | Yes |
|  |  |  |
| Null Keys and Null values | One null key ,Any null values | Not permit null keys and values |
|  |  |  |
| Iterator type | Fail fast iterator | Fail safe iterator |
|  |  |  |
| Performance | Fast | Slow in comparision |
|  |  |  |
| Superclass and Legacy | AbstractMap , No | Dictionary , Yes |

|  |  |
| --- | --- |
| HashMap | LinkedHashMap |
| Implements Map interface | Implements Map interface |
| Not synchronized | Not synchronized |
| Iterator returned by it is fail-fast nature | Iterator returned by it is fail-fast nature |
| Allows one null key and many null values | Allows one null key and many null values |
| Doesnot allow duplicate keys | Doesnot allow duplicate keys |
| Doesnot maintain any ordering of keys/values | Maintain insertion order of keys |
| Requires less memory | Requires more memory as in entry array chain it maintains two extra pointer before and after to keep track of the pre and next entries |
| Extends AbstractMap | Extends HashMap |
| It has better performance | It has somewhat slow performance as it maintains doubly linked list |

|  |  |
| --- | --- |
| HashMap | TreeMap |
| Implements Map interface | Implements Map interface |
| Not synchronized | Not synchronized |
| Iterator returned by it is fail-fast nature | Iterator returned by it is fail-fast nature |
| Doesnot allow duplicate keys | Doesnot allow duplicate keys |
| It is implemented as HashTable | Implemented as RedBlack Tree |
| Does not maintain key ordering | Sorted according to key |
| Implements Map interface | Implements Map,NavigableMap and SortedMap |
| It has better performance. Put/get/remove is of O(constant) | It has somewhat slow performance as it maintains key ordering. Put/get/remove is of O(logn) |
| Allow one null key and many null values | Won’t allow null key many null values |

**What is Fail-First and Fail-Safe :🡪**

Iterator in the HashMap/HashTable/TreeMap/LinkedHashMap is fail-first. This means if we want to add/remove any element (except using iterator’s remove method)in the Map while iterating through the map then it will throw **ConcurrentModificationException.**

While the enumeration in the HashTable is not fail-first i.e while iterating through the enumeration we can modify the table itself by adding/removing entries.

# Fail Fast Vs Fail Safe Iterator In Java : Java Developer Interview Questions

Difference between Fail fast and fail safe iterator  or  Fail fast vs Fail Safe iterator is one of those questions which are  used to test your knowledge about the topic Concurrency.  
Before we discuss in detail about fail safe iterator and fail fast iterator in addition to  their comparison , we should understand the term*Concurrent Modification*.  
  
**What is Concurrent Modification ?**  
  
When one or more thread is iterating over the collection, in between, one thread changes the structure of the collection (either adding the element to the collection or by deleting the element in the collection or by updating the value at particular position in the collection) is known as Concurrent Modification  
  
**Difference between Fail Fast iterator and Fail Safe iterator**  
  
**Fail fast Iterator**  
  
Fail fast iterator while iterating through the collection , instantly throws Concurrent Modification Exception if there is structural modification  of the collection . Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.   
  
Fail-fast iterator can throw ConcurrentModificationException in two scenarios :

*Single Threaded Environment*  
   
After the creation of the iterator , structure is modified at any time by any method other than iterator's own remove method.   
   
*Multiple Threaded Environment*  
  
 If one thread is modifying the structure of the collection while other thread is iterating over it .  
  
  
According to  [Oracle docs](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) , **the fail-fast behavior of an iterator cannot be guaranteed** as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness:**the fail-fast behavior of iterators should be used only to detect bugs.**  
  
  
**Interviewer : How  Fail  Fast Iterator  come to know that the internal structure is modified ?**  
Iterator read internal data structure (object array) directly . The internal data structure(i.e object array) should not be modified while iterating through the collection. To ensure this it maintains an internal  flag *"mods" .*Iterator checks the *"mods" flag*whenever it gets the next value (using hasNext() method and next() method). Value of*mods* flag changes whenever there is an structural modification. Thus indicating iterator to throw ConcurrentModificationException.  
  
  
**Fail Safe Iterator :**  
  
Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure.Any structural modification done to the iterator affects the copied data structure.  So , original data structure remains  structurally unchanged .Hence , no ConcurrentModificationException throws by the fail safe iterator.  
  
Two  issues associated with Fail Safe Iterator are :  
  
1. Overhead of maintaining the copied data structure i.e memory.  
  
2.  Fail safe iterator does not guarantee that the data being read is the data currently in the original data structure.   
  
According to [Oracle docs](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) , fail safe iterator is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don’t want to synchronize traversals, yet need to preclude interference among concurrent threads. The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This **array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException**.The iterator will not reflect additions, removals, or changes to the list since the iterator was created. Element-changing operations on iterators themselves (remove(), set(), and add()) are not supported. These methods throw UnsupportedOperationException.  
  
  
  
**Example of Fail Fast Iterator and Fail Safe Iterator**

**import** **java.util.HashMap**;

**import** **java.util.Iterator**;

**import** **java.util.Map**;

**public** **class** **FailFastExample**

{

**public** **static** **void** **main**(String[] args)

{

Map<String,String> premiumPhone = **new** HashMap<String,String>();

premiumPhone.put("Apple", "iPhone");

premiumPhone.put("HTC", "HTC one");

premiumPhone.put("Samsung","S5");

Iterator iterator = premiumPhone.keySet().iterator();

**while** (iterator.hasNext())

{

System.out.println(premiumPhone.get(iterator.next()));

premiumPhone.put("Sony", "Xperia Z");

}

}

}

**Output :**

iPhone

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.HashMap$HashIterator.nextEntry(Unknown Source)

at java.util.HashMap$KeyIterator.next(Unknown Source)

at FailFastExample.main(FailFastExample.java:**20**)

**Fail Safe Iterator Example :**

**import** **java.util.concurrent.ConcurrentHashMap**;

**import** **java.util.Iterator**;

**public** **class** **FailSafeExample**

{

**public** **static** **void** **main**(String[] args)

{

ConcurrentHashMap<String,String> premiumPhone =

**new** ConcurrentHashMap<String,String>();

premiumPhone.put("Apple", "iPhone");

premiumPhone.put("HTC", "HTC one");

premiumPhone.put("Samsung","S5");

Iterator iterator = premiumPhone.keySet().iterator();

**while** (iterator.hasNext())

{

System.out.println(premiumPhone.get(iterator.next()));

premiumPhone.put("Sony", "Xperia Z");

}

}

}

**Output :**

S5

HTC one

iPhone

**Recap : Difference between Fail Fast Iterator and Fail Safe Iterator**

|  |  |  |
| --- | --- | --- |
|  | **Fail Fast Iterator** | **Fail Safe Iterator** |
| Throw ConcurrentModification Exception | Yes | No |
| Clone object | No | Yes |
| Memory Overhead | No | Yes |
| Examples | HashMap,Vector,ArrayList,HashSet | CopyOnWriteArrayList, ConcurrentHashMap |

**ConcurrentHashMap**

**HashMap vs ConcurrentHashMap**

**1.  Thread -Safe :**  
  
ConcurrentHashMap is thread-safe that is the code can be accessed by multiple thread at a time .      
     while HashMap is not thread-safe .  
  
**2.  Synchronization Method :**  
  
 HashMap can be synchronized by using      
    synchronizedMap(HashMap)  method .  By using this    
    method we get a HashMap object which is equivalent   
    to the HashTable object . So every modification  is performed      
    on  Map is locked on Map object.

**import** **java.util.\***;

**public** **class** **HashMapSynchronization** {

**public** **static** **void** **main**(String[] args) {

// create map

Map<String,String> map = **new** HashMap<String,String>();

// populate the map

map.put("1","ALIVE ");

map.put("2","IS");

map.put("3","AWESOME");

// create a synchronized map

Map<String,String> syncMap = Collections.synchronizedMap(map);

System.out.println("Synchronized map :"+syncMap);

}

}

   ConcurrentHashMap synchronizes or locks on the certain portion of the Map . To optimize  
   the performance of ConcurrentHashMap , Map is divided into different partitions depending  
   upon the Concurrency level . So that we do not need to synchronize the whole Map Object.  
  
  
**3.  Null Key**  
  
  
     ConcurrentHashMap does not allow NULL values . So the key can not be null in  
     ConcurrentHashMap .While In HashMap there can only be one null key .  
  
   
**4.  Performance**   
  
     In single threaded environment HashMap is usually faster than ConcurrentHashMap . As      
     only single thread can access the certain portion of the Map and thus reducing the performance .   
     While in HashMap any number of threads can access the code at the same time .

# [How to use ConcurrentHashMap in Java - Example Tutorial and Working](http://javarevisited.blogspot.in/2013/02/concurrenthashmap-in-java-example-tutorial-working.html)

ConcurrentHashMap in Java is introduced as an alternative of Hashtable in Java 1.5 as part of Java concurrency package. Prior to Java 1.5 if you need a Map implementation, which can be safely used in a concurrent and multi-threaded Java program, than, you only have [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) or [synchronized Map](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) because HashMap is not [thread-safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html). With ConcurrentHashMap, now you have better choice; because, not only it can be safely used in concurrent multi-threaded environment but also provides better performance over Hashtable and synchronizedMap. ConcurrentHashMap performs better than earlier two because it only locks a portion of Map, instead of whole Map, which is the case with [Hashtable and synchronized Map](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html). CHM allows concurred read operations and same time, maintains integrity by synchronizing write operations. We have seen basics of ConcurrentHashMap on [Top 5 Java Concurrent Collections from JDK 5 and 6](http://javarevisited.blogspot.sg/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html) and in this Java tutorial, we will learn :

       How ConcurrentHashMap works in Java or how it is implemented in Java.

       When to use ConcurrentHashMap in Java

       ConcurrentHashMap examples in Java

       And some important properties of CHM .

## How ConcurrentHashMap is implemented in Java

ConcurrentHashMap is introduced as an alternative of Hashtable and provided all functions supported by Hashtable with additional feature called "concurrency level", which allows ConcurrentHashMap to partition Map. ConcurrentHashMap allows multiple readers to read concurrently without any [blocking](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html). This is achieved by partitioning Map into different parts based on concurrency level and locking only a portion of Map during updates. Default concurrency level is 16, and accordingly Map is divided into 16 part and each part is governed with different lock. This means, 16 thread can operate on Map simultaneously, until they are operating on different part of Map. This makes ConcurrentHashMap high performance despite keeping thread-safety intact.  Though, it comes with caveat. Since update operations like put(), remove(), putAll() or clear() is not [synchronized](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), **concurrent retrieval may not reflect most recent change on Map**.

In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries. Another important point to remember is iteration over CHM, [Iterator](http://javarevisited.blogspot.com/2011/10/java-iterator-tutorial-example-list.html) returned by keySet of ConcurrentHashMap are weekly consistent and they only reflect state of ConcurrentHashMap and certain point and may not reflect any recent change. Iterator of ConcurrentHashMap's keySet area also [fail-safe](http://javarevisited.blogspot.in/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and doesn’t throwConcurrentModificationExceptoin..

Default concurrency level is 16 and can be changed, by providing a number which make sense and work for you while creating ConcurrentHashMap. Since concurrency level is used for internal sizing and indicate number of concurrent update without contention, so, if you just have few writers or thread to update Map keeping it low is much better. ConcurrentHashMap also uses ReentrantLock to internally lock its segments.

## ConcurrentHashMap putifAbsent example in Java

ConcurrentHashMap examples are similar to [Hashtable examples](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html), we have seen earlier,  but worth knowing is use of putIfAbsent() method. Many times we need to insert entry into Map, if its not present already, and we wrote following kind of code:

synchronized(map){

**if** (map**.**get(key) **==** *null*){

**return** map**.**put(key, value);

  } **else**{

**return** map**.**get(key);

  }

}

Though this code will work fine in [HashMap and Hashtable](http://java67.blogspot.sg/2012/08/5-difference-between-hashtable-hashmap-Java-collection.html), This won't work in ConcurrentHashMap; because, during put operation whole map is not locked, and while one thread is putting value, other thread's get() call can still return null which result in one thread overriding value inserted by other thread. Ofcourse, you can wrap whole code in [synchronized block](http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) and make it [thread-safe](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) but that will only make your code single threaded. ConcurrentHashMapprovides putIfAbsent(key, value) which does same thing but atomically and thus eliminates above race condition.

## When to use ConcurrentHashMap in Java

[Java ConcurrentHashMap Example Tutorial and internal working](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)

ConcurrentHashMap is best suited when you have multiple readers and few writers. If writers outnumber reader, or writer is equal to reader, than performance of ConcurrentHashMap effectively reduces to [synchronized map](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) or [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html). Performance of CHM drops, because you got to lock all portion of Map, and effectively each reader will wait for another writer, operating on that portion of Map. ConcurrentHashMap is a good choice for caches, which can be initialized during application start up and later accessed my many request processing threads. As javadoc states, CHM is also a [good replacement of Hashtable](http://javarevisited.blogspot.sg/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html) and should be used whenever possible, keeping in mind, that CHM provides slightly weeker form of synchronization than Hashtable.

### Summary

Now we know What is ConcurrentHashMap in Java and when to use ConcurrentHashMap, it’s time to know and revise some important points about CHM in Java.

1. ConcurrentHashMap allows concurrent read and thread-safe update operation.

2. During update operation, ConcurrentHashMap only lock a portion of Map instead of whole Map.

3. Concurrent update is achieved by internally dividing Map into small portion which is defined by concurrency level.

4. Choose concurrency level carefully as a significant higher number can be waste of time and space and lower number may introduce thread contention in case writers over number concurrency level.

5. All operations of ConcurrentHashMap are [thread-safe](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html).

6. Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.

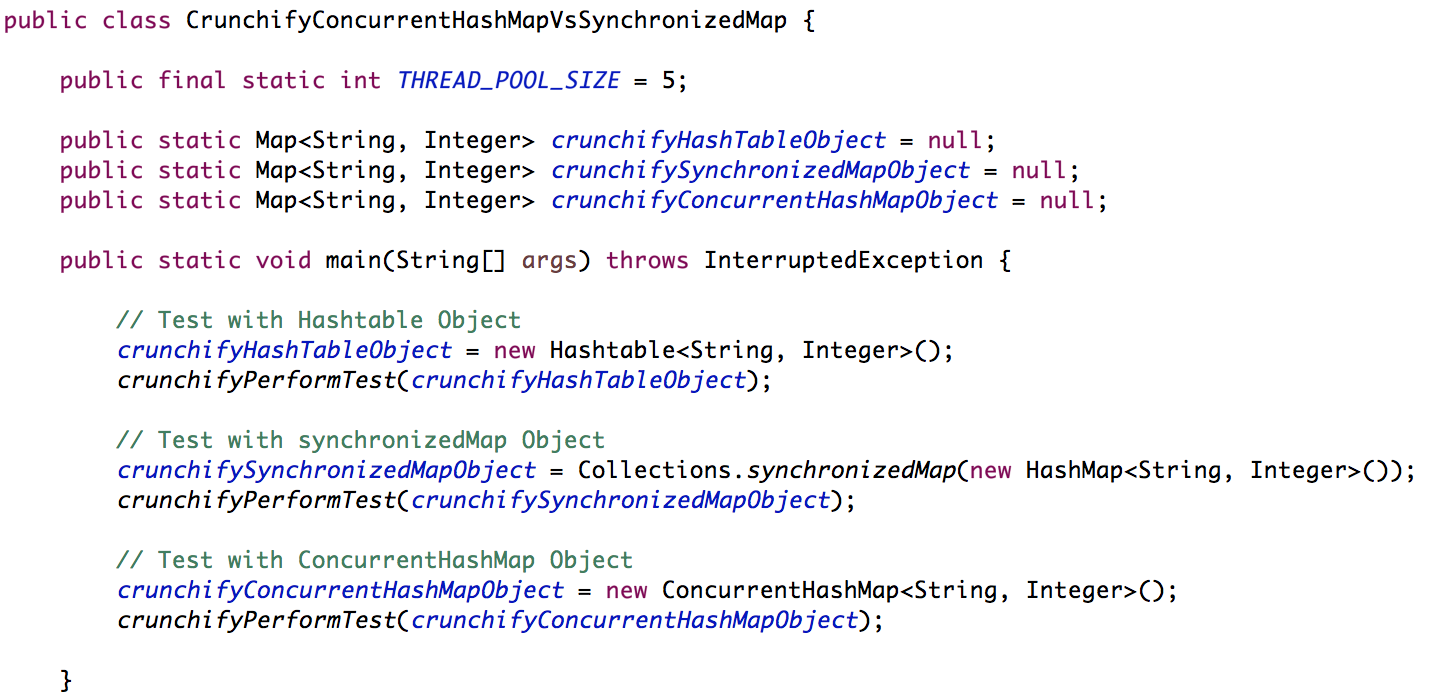
7. Iterator returned by ConcurrentHashMap is weekly consistent, [fail safe](http://javarevisited.blogspot.com/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and never throw ConcurrentModificationException. In Java.

8. ConcurrentHashMap doesn't allow null as key or value.

9. You can use ConcurrentHashMap in place of [Hashtable](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html) but with caution as CHM doesn't lock whole Map.

10. During putAll() and clear() operations, concurrent read may only reflect insertion or deletion of some entries.

That’s all on **What is ConcurrentHashMap in Java** and when to use it. We have also seen little bit about internal working of ConcurrentHashMap and how it achieves it’s thread-safety and better performance over Hashtable and synchronized Map. Use ConcurrentHashMap in Java program, when there will be more reader than writers and it’s a good choice for creating cache in Java as well.

[](http://cdn.crunchify.com/wp-content/uploads/2015/01/Crunchify-ConcurrentHashMap-Vs.-SynchronizedMap-Example.png)HashMap is a very powerful data structure in [Java](http://crunchify.com/category/java-web-development-tutorial/). We use it everyday and almost in all applications. There are quite a few examples which I have written before on [How to Implement Threadsafe cache](http://crunchify.com/implement-simple-threadsafe-cache-using-hashmap-without-using-synchronized-collection/), How to convert [Hashmap to Arraylist](http://crunchify.com/how-to-convert-hashmap-to-arraylist-in-java/)?

We used Hashmap in both above examples but those are pretty simple use cases of Hashmap. HashMap is a non-synchronized collection class.

Do you have any of below questions?

* What’s the difference between ConcurrentHashMap and Collections.synchronizedMap(Map)?
* What’s the difference between ConcurrentHashMap and Collections.synchronizedMap(Map) in term of performance?
* ConcurrentHashMap vs Collections.synchronizedMap()
* Popular HashMap and ConcurrentHashMap interview questions

In this tutorial we will go over all above queries and reason why and how we could Synchronize Hashmap?

Why?

The Map object is an associative containers that store elements, formed by a combination of a uniquely identify key and a mapped value. If you have very highly concurrent application in which you may want to modify or read key value in different threads then it’s ideal to use Concurrent Hashmap. Best example is [Producer Consumer](http://crunchify.com/java-producer-consumer-example-handle-concurrent-read-write/) which handles concurrent read/write.

So what does the thread-safe Map means? If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally to avoid an inconsistent view of the contents.

How?

There are two ways we could synchronized HashMap

1. Java Collections synchronizedMap() method
2. Use ConcurrentHashMap

HashMap Vs. synchronizedMap Vs. ConcurrentHashMap

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //Hashtable  Map<String, String> normalMap = new Hashtable<String, String>();    //synchronizedMap  synchronizedHashMap = Collections.synchronizedMap(new HashMap<String, String>());    //ConcurrentHashMap  concurrentHashMap = new ConcurrentHashMap<String, String>(); |

ConcurrentHashMap

* You should use ConcurrentHashMap when you need very high concurrency in your project.
* It is thread safe without synchronizing the whole map.
* Reads can happen very fast while write is done with a lock.
* There is no locking at the object level.
* The locking is at a much finer granularity at a hashmap bucket level.
* ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
* ConcurrentHashMap uses multitude of locks.

SynchronizedHashMap

* Synchronization at Object level.
* Every read/write operation needs to acquire lock.
* Locking the entire collection is a performance overhead.
* This essentially gives access to only one thread to the entire map & blocks all the other threads.
* It may cause contention.
* SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

Now let’s take a look at code

1. Create class CrunchifyConcurrentHashMapVsSynchronizedHashMap.java
2. Create object for each HashTable, SynchronizedMap and CrunchifyConcurrentHashMap
3. Add and retrieve 500k entries from Map
4. Measure start and end time and display time in milliseconds
5. We will use [ExecutorService](http://crunchify.com/how-to-run-multiple-threads-concurrently-in-java-executorservice-approach/) to run 5 threads in parallel

CrunchifyConcurrentHashMapVsSynchronizedMap.java

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | package crunchify.com.tutorials;    import java.util.Collections;  import java.util.HashMap;  import java.util.Hashtable;  import java.util.Map;  import java.util.concurrent.ConcurrentHashMap;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  import java.util.concurrent.TimeUnit;    /\*\*  \* @author Crunchify.com  \*  \*/    public class CrunchifyConcurrentHashMapVsSynchronizedMap {    public final static int THREAD\_POOL\_SIZE = 5;    public static Map<String, Integer> crunchifyHashTableObject = null;  public static Map<String, Integer> crunchifySynchronizedMapObject = null;  public static Map<String, Integer> crunchifyConcurrentHashMapObject = null;    public static void main(String[] args) throws InterruptedException {    // Test with Hashtable Object  crunchifyHashTableObject = new Hashtable<String, Integer>();  crunchifyPerformTest(crunchifyHashTableObject);    // Test with synchronizedMap Object  crunchifySynchronizedMapObject = Collections.synchronizedMap(new HashMap<String, Integer>());  crunchifyPerformTest(crunchifySynchronizedMapObject);    // Test with ConcurrentHashMap Object  crunchifyConcurrentHashMapObject = new ConcurrentHashMap<String, Integer>();  crunchifyPerformTest(crunchifyConcurrentHashMapObject);    }    public static void crunchifyPerformTest(final Map<String, Integer> crunchifyThreads) throws InterruptedException {    System.out.println("Test started for: " + crunchifyThreads.getClass());  long averageTime = 0;  for (int i = 0; i < 5; i++) {    long startTime = System.nanoTime();  ExecutorService crunchifyExServer = Executors.newFixedThreadPool(THREAD\_POOL\_SIZE);    for (int j = 0; j < THREAD\_POOL\_SIZE; j++) {  crunchifyExServer.execute(new Runnable() {  @SuppressWarnings("unused")  @Override  public void run() {    for (int i = 0; i < 500000; i++) {  Integer crunchifyRandomNumber = (int) Math.ceil(Math.random() \* 550000);    // Retrieve value. We are not using it anywhere  Integer crunchifyValue = crunchifyThreads.get(String.valueOf(crunchifyRandomNumber));    // Put value  crunchifyThreads.put(String.valueOf(crunchifyRandomNumber), crunchifyRandomNumber);  }  }  });  }    // Make sure executor stops  crunchifyExServer.shutdown();    // Blocks until all tasks have completed execution after a shutdown request  crunchifyExServer.awaitTermination(Long.MAX\_VALUE, TimeUnit.DAYS);    long entTime = System.nanoTime();  long totalTime = (entTime - startTime) / 1000000L;  averageTime += totalTime;  System.out.println("500K entried added/retrieved in " + totalTime + " ms");  }  System.out.println("For " + crunchifyThreads.getClass() + " the average time is " + averageTime / 5 + " ms\n");  }  } |

Result

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | Test started for: class java.util.Hashtable  500K entried added/retrieved in 1432 ms  500K entried added/retrieved in 1425 ms  500K entried added/retrieved in 1373 ms  500K entried added/retrieved in 1369 ms  500K entried added/retrieved in 1438 ms  For class java.util.Hashtable the average time 1407 ms    Test started for: class java.util.Collections$SynchronizedMap  500K entried added/retrieved in 1431 ms  500K entried added/retrieved in 1460 ms  500K entried added/retrieved in 1387 ms  500K entried added/retrieved in 1456 ms  500K entried added/retrieved in 1406 ms  For class java.util.Collections$SynchronizedMap the average time 1428 ms    Test started for: class java.util.concurrent.ConcurrentHashMap  500K entried added/retrieved in 413 ms  500K entried added/retrieved in 351 ms  500K entried added/retrieved in 427 ms  500K entried added/retrieved in 337 ms  500K entried added/retrieved in 339 ms  For class java.util.concurrent.ConcurrentHashMap the average time 373 ms  <== Much faster |

**Interviewer : Why we need ConcurrentHashMap when we already had Hashtable ?**  
  
Hashtable provides concurrent access to the Map.Entries objects by locking the entire map to perform any sort of operation (update,delete,read,create). Suppose we have a web application , the overhead created by Hashtable  (locking the entire map) can be ignored under normal load. But under heavy load , the overhead of locking the entire map may prove fatal and may lead to delay response time and   overtaxing of the server.

This  is where ConcurrentHashMap comes to rescue. According to[ConcurrentHashMap Oracle docs,](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html)  
ConcurrentHashMap class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details. So the main purpose of this class is to provide the same functionality as of Hashtable but with a performance comparable to HashMap.   
  
ConcurrentHashMap achieves this by a simple tweak. So this leads to our main question

**How ConcurrentHashMap works in Java**  
  
According to ConcurrentHashMap Oracle docs,   
  
The constructor of ConcurrentHashMap looks like this :  
  
**public ConcurrentHashMap*(int initialCapacity, float loadFactor, int concurrencyLevel)***  
  
So the above line  creates a new, empty map with the specified initial capacity, load factor and concurrency level.

where,

**Important Parameters to consider from ConcurrentHashMap Constructor:**

**initialCapacity** - the initial capacity. The implementation performs *internal sizing to accommodate this many elements.*  
**concurrencyLevel** - the estimated number of concurrently updating threads. The implementation performs *internal sizing to try to accommodate this many threads.*

In the [ConcurrentHashMap Api](http://www.docjar.com/html/api/java/util/concurrent/ConcurrentHashMap.java.html) , you will find the following constants.

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

static final int DEFAULT\_CONCURRENCY\_LEVEL = 16;

initial capacity parameter and concurrency level parameters of ConcurrentHashMap constructor (or Object) are  set to 16 by default.  
  
Thus, instead of a map wide lock, ConcurrentHashMap maintains  a list of 16 locks by default ( number of locks equal to the initial capacity , which is by default  16) each of which is used to lock on a single bucket of the Map.This indicates that 16 threads (number of threads equal to the concurrency level , which is by  default 16) can modify the collection at the same time , given ,each thread works on different bucket. So unlike hashtable, we perform any sort of operation ( update ,delete ,read ,create) without locking on entire map in ConcurrentHashMap.  
  
Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently *completed* update operations holding upon their onset.   
  
The allowed concurrency among update operations is guided by the optional concurrencyLevel constructor argument (default 16), which is used as a hint for internal sizing. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. Because placement in hash tables is essentially random, the actual concurrency will vary. Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention  
  
**Interviewer : Can two threads update the ConcurrentHashMap simultaneously ?**  
  
                                                       
Yes it is possible that two threads can simultaneously write on the ConcurrentHashMap. ConcurrentHashMap default implementation allows 16 threads to read and write in parallel.   
But in the worst case scenario , when two objects lie in the same segment or same partition, then parallel write would not be possible.  
  
**Interviewer : Why ConcurrentHashMap does not allow null keys and null values ?**  
  
According to the [author of the ConcurrentHashMap (Doug lea himself)](http://cs.oswego.edu/pipermail/concurrency-interest/2006-May/002485.html)

The main reason that nulls aren't allowed in ConcurrentMaps (ConcurrentHashMaps, ConcurrentSkipListMaps) is that ambiguities that may be just barely tolerable in non-concurrent maps can't be accommodated. The main one is that if map.get(key) returns null, you can't detect whether the key explicitly maps to null vs the key isn't mapped. In a non-concurrent map, you can check this via map.contains(key), but in a concurrent one, the map might have changed between   
calls.  
  
In simple words,   
  
The code is like this :   
 

**if** (map.containsKey(k)) {

**return** map.get(k);

} **else** {

**throw** **new** **KeyNotPresentException**();

}

It might be possible that key k might be deleted in between the get(k) and containsKey(k) calls. As a result , the code will return null as opposed to KeyNotPresentException (Expected Result if key is not present).   
  
**Interviewer : What is the difference between HashMap and ConcurrentHashMap?**  
  
The HashMap was not thread safe and therefore could not be utilized in multi-threaded applications.  The ConcurrentHashMap was introduced to overcome this shortcoming and also as an alternative to using HashTable and synchronized Maps for greater performance and uses the standard Hashing algorithms to generate hash code for storing the key value pairs.For more difference between HashMap and ConcurrentHashMap check this [popular interview question HashMap vs ConcurrentHashMap in java](http://javahungry.blogspot.com/2014/02/hashmap-vs-concurrenthashmap-java-collections-interview-question.html).  
  
  
  
**Interviewer : Can multiple threads read from the Hashtable concurrently ?**  
  
No multiple threads can not read simultaneously from Hashtable. Reason, the get() method of  Hashtable is synchronized. As a result , at a time only one thread can access the get() method .  
It is possible to achieve full  concurrency for reads (all the threads read at the same time) in  ConcurrentHashMap by using volatile keyword.  
  
**Interviewer: Does ConcurrentHashMap Iterator behaves like fail fast iterator or fail safe Iterator?**   
  
ConcurrentHashMap iterator behaves like fail safe iterator. It will not throw ConcurrentModificationException . We have already discussed [Fail Fast Iterator vs Fail Safe Iterator](http://javahungry.blogspot.com/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html).  
  
  
**Interviewer : Why does Java provide default value of partition count as 16 instead of very high value ?**  
  
According to Java docs ,  
  
Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention.  
  
  
  
  
**Interviewer : Can you write the simple  example which proves ConcurrentHashMap class behaves like fail safe iterator?**  
  
  
**ConcurrentHashMap Example :**

**import** **java.util.concurrent.ConcurrentHashMap**;

**import** **java.util.Iterator**;

**public** **class** **ConcurrentHashMapExample**

{

**public** **static** **void** **main**(String[] args)

{

ConcurrentHashMap<String,String> premiumPhone = **new** ConcurrentHashMap<String,String>();

premiumPhone.put("Apple", "iPhone6");

premiumPhone.put("HTC", "HTC one");

premiumPhone.put("Samsung","S6");

Iterator iterator = premiumPhone.keySet().iterator();

**while** (iterator.hasNext())

{

System.out.println(premiumPhone.get(iterator.next()));

premiumPhone.put("Sony", "Xperia Z");

}

}

}

**Output :**

S6

HTC one

iPhone6